



# File Format Documentation

For BRW v4.x, BXR v3.x and BCMP v1.x  
(available on BrainWave v5.x)

## Table of Contents

Introduction .....	<b>3</b>
Overview .....	<b>4</b>
Formatting Conventions .....	4
HDF Objects .....	4
JSON Objects .....	4
Well Notation .....	6
Channel Notation .....	6
Frames .....	6
Data Formats .....	<b>8</b>
BRW- and BXR- Files .....	<b>9</b>
Common Structure .....	9
Root .....	9
Attributes .....	9
Contained Objects .....	10
TOC .....	10
BRW-File .....	12
Structure .....	12
Contained Objects .....	12
Raw .....	13
Event Based Sparse Raw .....	14
Wavelet Based Encoded Raw .....	14
BXR-File .....	20
Structure .....	20
Contained Objects .....	20
BCMP-File .....	<b>23</b>
Contained Objects .....	23



# Introduction

Welcome to the BrainWave user documentation!

This document provides information on the BrainWave data file formats based on Hierarchical Data Format 5 (HDF5), namely:

- BRW-File (**3**Brain **RaW** data file) version 4.0,
- BXR-File (**3**Brain **eX**periment **R**esults file) version 3.0
- BCMP-File (**3**Brain **CoMP**ound data file) version 1.0.

These files are collectively referred to in this document as BrainWave Files.

BrainWave Files are used for accessing and storing information from experiments performed with 3Brain's proprietary BioSPU-microchips based cell-electronic interface (CEI) plates (HD-MEA or CorePlate™), collectively referred to as “3Brain CEI plates”.

BrainWave Files are pure HDF5 files with a .brw, .bxx or .bcmp extension and with a custom data organization and metadata information and allow for reading/append/writing operations in BrainWave software (5.0 or higher). All files are written with the HDF5 version 1.12.1.

HDF5 is a file format designed to store large and complex data collections that are organized in a filesystem-like structure and that can be accessed with a POSIX-like syntax */path/to/resource*. HDF5 has no limit on the number and size of data objects, is completely portable, can be used even on massive parallel systems and comes with an API for C, C++, Fortran 90, Java and CLI .NET. More information on HDF5, its data structure, tools and software and example programs to read and write to it can be found on <http://www.hdfgroup.org/>. On the same website it is also possible to download the HDFView Software, a simple tool that allows to open HDF5 files, and thus also BrainWave Files, and to browse their content.

This document describes data organization inside BrainWave HDF5-based BRW-, BXR-Files, and BCMP-Files.







# Overview

## Formatting Conventions

The following formatting conventions are used in this document:

- A paragraph corresponding to a versionable object has a section for each available version, such as `Version 200`.

In file-tree graphic representation, Groups are indicated with grey-filled rectangles (  ), Datasets with cyan-filled rectangles (  ), Json-encoded objects with a green-filled rectangles (  ) and Attributes with cyan-bordered, white-filled rectangles (  ).

## HDF Objects

The following HDF objects are used to define the structures of BrainWave Files:

- Group: a collection of objects such as datasets and other groups
- Dataset: a 1- or multi-dimension array of data elements storing numerical data
- Attribute: metadata information associated to a group or a dataset

Groups, which can be seen as the folders in a filesystem, can be either simple groups containing non-versionable objects or versionable groups containing objects whose names, layout and organization depend on the group's version. Versionable groups have at least one Attribute, named Version, which defines the version for their content and might have additional attributes defining optional layout information for their content. The file root group ('/'; hereafter referred to as Root) is also a versionable group having a Version Attribute defining the general version of the file.

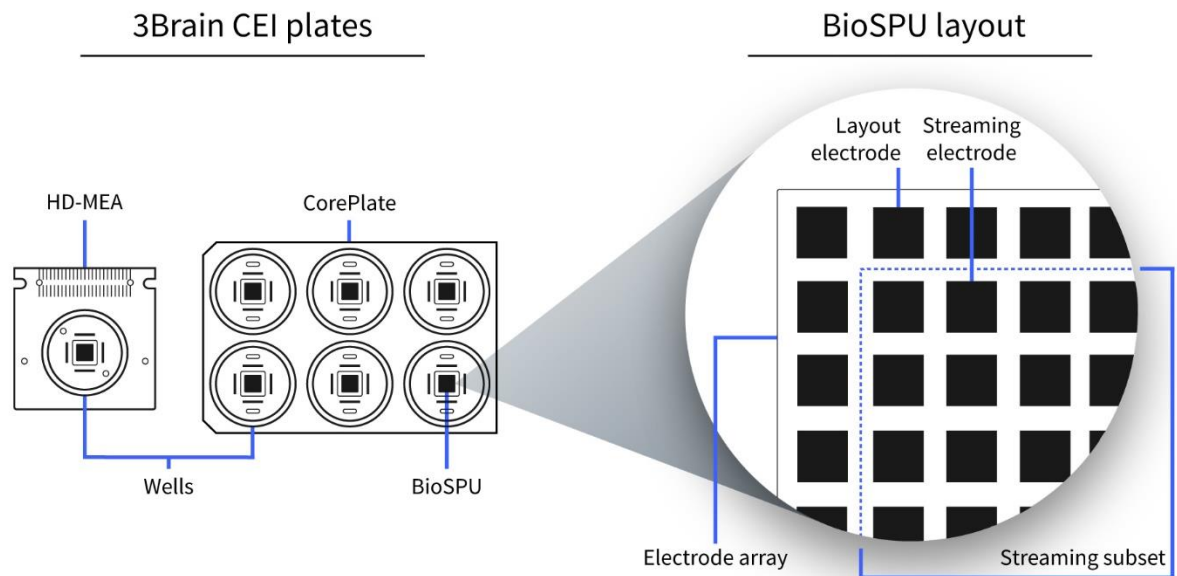
In addition to reading the information contained in BrainWave Files, some users might also be interested in writing to them by adding their specific data. In this case it is recommended to add data in separate groups (folders) from those used by BrainWave to reduce the chances of conflicts with the 3Brain's data structure, which might compromise the correct loading of the files in BrainWave.

## JSON Objects

Some information contained in the BrainWave Files is stored in JSON-formatted strings. More information on JSON can be found on <https://www.json.org/json-en.html>.

All JSON-Objects are versionable and contain a version value (`JsonVersion`) and, where needed, a value indicating the object type (`$type`).

## Definitions



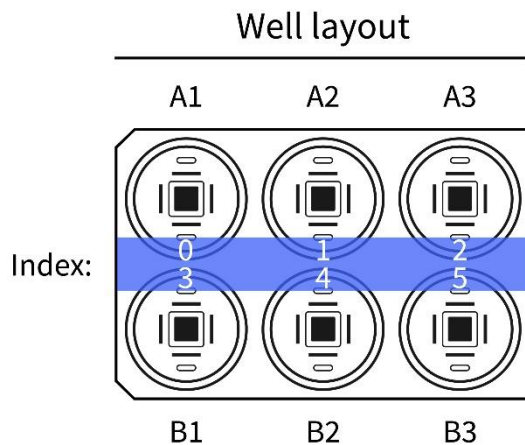
3Brain’s products make use of sophisticated culture plates to measure cells and establish a bidirectional communication with them. Such cell-electronic interface (CEI) plates, herein referred to as CEI Plates or simply Plates, integrate different numbers of culture chambers termed wells. HD-MEA are single-well CEI Plates, while CorePlate™ contains multiple wells. Each well incorporates at its base a biosignal processing unit (BioSPU)-microchip that integrates an array of sensing and actuating electrodes. The term “electrode” and “channel” are interchangeable in this document.

The BioSPU-microchip integrates a certain number of electrodes, always the same for each of the microchips within the same CEI-Plate, that are defined by the microchip layout and therefore are termed Layout Electrodes or Layout Channels.

However, when acquiring data only a subset of the layout channels may be streamed from the microchip and eventually stored to the BrainWave Files. For instance, this is the case when the user applies a region-of-interest (ROI) to the Layout Channels. The channels that are effectively streamed from the microchip are termed Streaming Electrodes or Streaming Channels.

## Well Notation

A well inside a CEI-Plate can be represented either by a linear index or by an identifier (`WellId`). The identifier uses a letter-number pair where the letter represents the row of the well and number of the column, inside the plate. Supposing to have a 6 well CEI Plate with 2 rows and 3 columns of wells, the top-left well is “A1” and the bottom-right well is “B3”. The linear index is a 0-based indexing notation that assigns an index to the well in the range  $[0\ n-1]$ , where  $n$  is the number of wells, by ordering the wells from left-to-right and then from top-to-bottom. In our case, 0 is the A1 well, 3 the B1 well and 5 the B3 well.



## Channel Notation

A BioSPU-microchip channel can be represented either with a linear index or by means of its well, row and column subscripts. Both indexes and subscripts are assigned based on the position of the channel in the layout and not in the streaming subset. Linear indexes use 0-based notation, while subscripts use 1-based notation. When using linear indexes, both wells and channels inside the wells are ordered left-to-right and top-to-bottom.

Let's suppose to have 24576 channels in a 6-well CEI Plate from A1 to B3, each well containing 4096 channels, in a 64 x 64 grid. When using the subscript notation, the channels are identified by their well subscript, varying in the range  $[1\ 6]$ , and by their row and column subscripts, which both vary in the range  $[1\ 64]$ . Hence, the channel (1, 1, 1) is the channel in well A1, first row and first column, while the channel (6, 64, 64) is the channel in well B3, last row and last column. When using the linear index notation, a channel is defined by a value ranging in  $[0\ 24575]$ . The 0 index is the first channel at well A1, row 1 and column 1, the 1 index is the channel at well A1, row 1 and column 2, the index 64 is the channel at well A1, row 2 and column 1, the index 4096 is the channel at well A2, row 1 and column 1, the index 12288 is the channel at well B1, row 1 and column 1, and so on. Inside the file, channels are always referred to by their index (`ChIdx`).

## Frames

Recorded values from channels are acquired as a sequence of Frames, where each Frame contains all the samples of the streaming channels. A Frame corresponds also to a certain time instance,

i.e. the time at which the frame was captured. Time instants in BrainWave Files are stored as discrete 64-bit integers. Frames time positions can be converted back into seconds using the SamplingRate (see [Attributes](#)) used to record the data:

$$\text{TimeInSeconds} = \text{Frame}/\text{SamplingRate}$$

## Samples

Samples recorded from channels are stored as digital values. To convert between a digital sampled value (DigitalValue) and the corresponding analog value in microvolt (AnalogValue) the following formula can be used:

$$\text{AnalogValue} = \text{MinAnalogValue} + \text{DigitalValue} * (\text{MaxAnalogValue} - \text{MinAnalogValue}) / (\text{MaxDigitalValue} - \text{MinDigitalValue})$$

Where MaxAnalogValue, MinAnalogValue, MaxDigitalValue and MinDigitalValue are values that are saved inside the BrainWave Files (see [Attributes](#)).

# Data Formats

BrainWave 5 uses three file types to store and organize data, BRW-, BXR- and BCMP-files. BRW- and BXR-File types use the open source HDF5 hierarchical data format ([hdfgroup.org](http://hdfgroup.org)) that supports most operating systems (including Windows, Linux and Mac OS) and most used data analysis environments, such as Python, Matlab, Scilab, Octave and R. BCMP-Files are instead JSON files.



**BRW** stands for **B**rainwave **RaW** data file and contains the Source data, i.e. continuous or quasi-continuous data streamed from BioCAM or HyperCAM platforms. For instance, BRW files can store raw data recorded from all the electrodes and all the wells contained in a CorePlate™ multiwell, or temporal subsets (recording intervals) of a full raw data stream.



**BXR** stands for **B**rainwave **eX**periment **R**esult data file and contains Results data, i.e. discrete data obtained by analyzing and processing the raw data. For instance, BXR files can contain the timestamps and the waveforms of the detected spikes from firing cells. Conceptually, a BXR-file is created starting from a BRW-file, hence a BXR-file is associated and linked to its parent BRW-file (whereas the opposite is not true). A BXR-file can be deleted and recreated starting from the BRW-file, if existing, while if a BRW-file is deleted the contained data are lost forever. However, since BRW-files are typically large files containing the fullest amount of data, sometimes it may be useful to delete the BRW-file once the relevant results have been obtained and saved in a BXR-file.



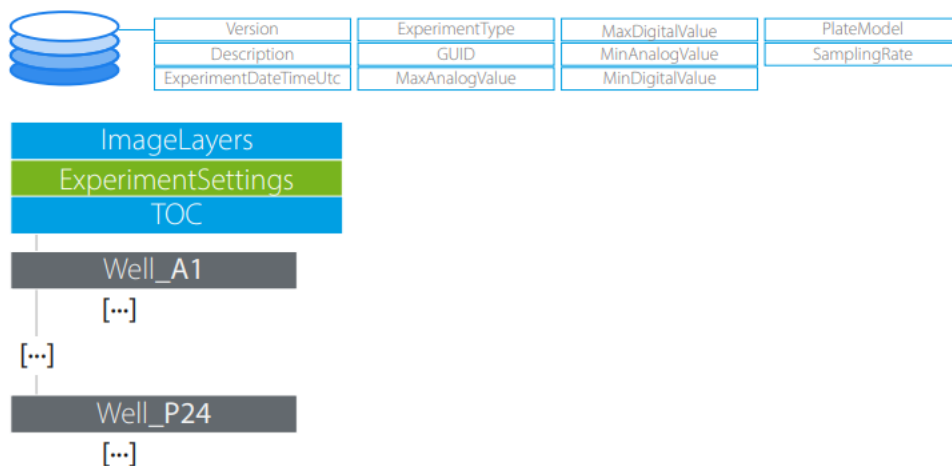
**BCMP** stands for **B**rainwave **Co**MPosite data file and represents a view of multiple experiments. The BCMP format contains links and instructions to combine together multiple BRW-files or BXR-files. For instance, this is useful when multiple BXR-files were obtained from the same HD-MEA at different time points (e.g. at different days). Combining the files together allows you to see the data trend over time. Conceptually, a BCMP-file is created starting from BXR-files, hence a BCMP-file is associated and linked with multiple BXR-files.



# BRW- and BXR- Files

## Common Structure

### ROOT



### ATTRIBUTES

All files have several attributes that contain the bare minimum information needed to open and display the data correctly. This information ensures to open the file even when ExperimentSettings (see **Contained Objects**) information is corrupted (e.g., due to manipulation of its content with third-party software).

- Version: a 32-bit Integer indicating the file's version.
- Description: a string description of the file containing a small description of the file type.
- ExperimentDateTimeUtc: a 64-bit Integer value containing the date of the experiment in UTC time (Coordinated Universal Time).
- ExperimentType: a 16-bit Integer containing the experiment type. Available values are 0 for Standard Experiments, and 1 for EVOS Experiments.
- GUID: a Global Unique Identifier string that uniquely identifies the file.
- MaxAnalogValue: a 64-bit floating-point value containing the maximum voltage of the recording amplifiers sampling range, in microvolt.
- MaxDigitalValue: a 64-bit floating-point value containing the quantization level corresponding to the maximum voltage of the recording amplifier's sampling range.
- MinAnalogValue: a 64-bit floating-point value containing the minimum voltage of the recording amplifiers sampling range, in microvolt.
- MinDigitalValue: a 64-bit floating-point value containing the quantization level corresponding to the minimum voltage of the recording amplifier's sampling range.

- `PlateModel`: a 16-bit Integer containing the 3Brain Plate Model ID.
- `SamplingRate`: a 64-bit floating-point value containing the sampling frequency used for recording data.
- `SourceGUID`: found only in BXR-Files, contains the GUID of the source file.

## CONTAINED OBJECTS

- `ExperimentSettings`: a JSON-encoded string containing all information related to the experiment and its conditions of recording. Contains also all settings defined for the analysis that may have been performed. It has a `Status` attribute indicating whether the object is corrupted or not. A value different from 0 means the settings are corrupted (this may have happened while manipulating the JSON string with third-party software).
- `TOC`: a Table of Contents as a 2-dimension dataset of 64-bit integers, containing the starting and ending frame of each data chunk that has been recorded (for details, see below).
- `ImageLayers`: a 1-dimension dataset of Bytes, containing all images added to the file and referenced inside the `ExperimentSettings` object. The first 4 bytes encode a 32-bit Integer for the number of images contained in the dataset. Then each image is preceded by 8 bytes encoding a 64-bit Integer for the image length. The image itself is encoded as a sequence of bytes as resulting from the `System.Drawing.ImageConverter.ConvertTo` method found inside .NET framework.
- `Well Groups`: versioned `WellGroup` HDF5 groups containing data recorded for each well, named with the format “`Well_WellId`” where `WellId` is the well identifier. In case of a single-well HD-MEA only one HDF5 group will be present, named `Well_A1`.

## TOC

BrainWave 5 stores data in chunks, which represent a small time slice. A Recording Interval is a usually larger time slice of the 3Brain Plate data stream that is recorded in a BrainWave File. A Recording Interval may be composed of one or multiple data chunks. To improve reading operations, information about the chunks and the Recording Interval are stored in a TOC object.

In all BRW- and BXR-Files the first two columns of the TOC are inside the Root, in the dataset “TOC”, a 2-dimension table of 64-bit Integers.

The TOC is made up by multiple rows and columns. The columns are contained in different datasets, therefore the TOC is split in different sub-TOCs. All sub-TOCs have the same number of rows. Each row in the table of content (shown below) describes a recorded chunk of data for a given Recording Interval.

The first column (FRAME START,  $FS(i)$ ) contains the Frame (included) at which the chunk starts, and the second one (FRAME END,  $FE(i)$ ) contains the Frame (excluded) at which the chunk ends. When chunks are within the same Recording Interval  $FS(i+1) = FE(i)$ . When instead  $FS(i+1) > FE(i)$ , it means a new Recording Interval was initiated at step  $i+1$ .

The other columns (`DATASET_1`, `DATASET_2` up to `DATASET_K` in the table below) are stored individually in 1-dimensional datasets in each `WellGroup`, one for each stored datatype, which is indicated in their name (e.g., `SpikeTOC` for spike data, `FpTOC` for field potential data). Each value inside these columns ( $DP_1(i)$ ,  $DP_2(i)$ ,  $DP_K(i)$ ) indicates the data position in the dataset of the corresponding datatype of the first recorded event for the chunk represented by the row. The

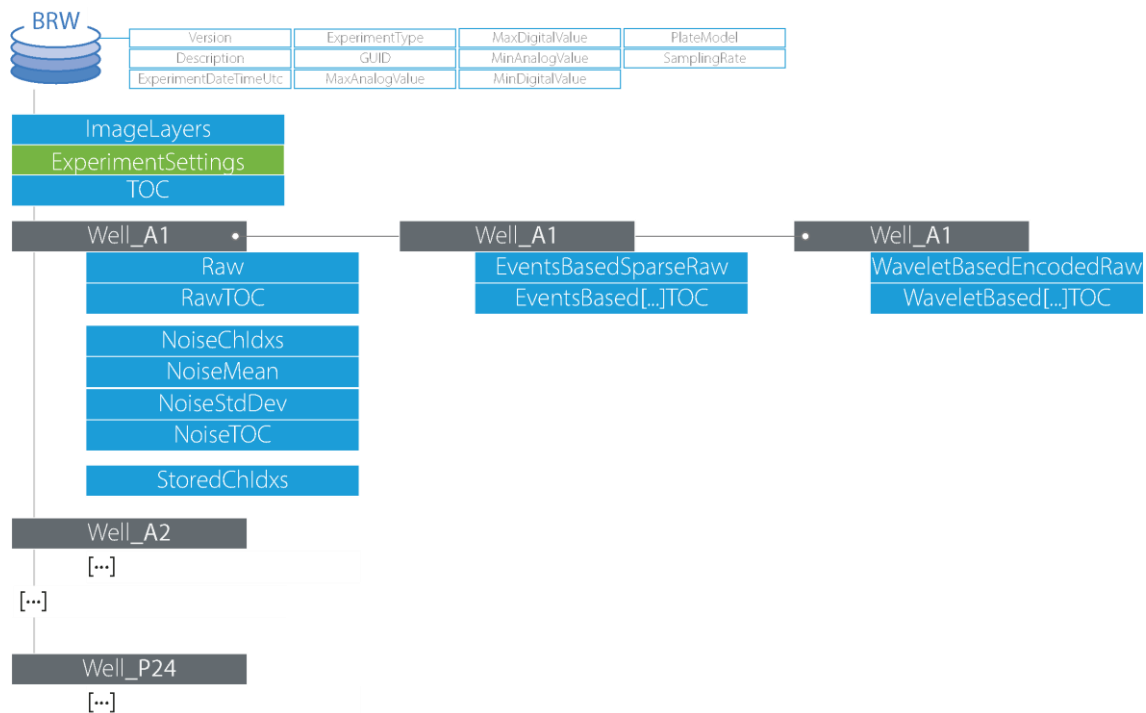
data position inside a dataset is the position in number of elements preceding that position. The range of values contained in the dataset for a desired chunk can then be obtained by taking all the values between the value indicated in this chunk's TOC row (included) and the value indicated in the next TOC row (excluded). For instance, let's suppose we are recording spike timestamp information inside the `SpikeTimes` dataset, a 1-dimensional array containing all the spike instants. Let's also suppose that at step  $i$  a new data chunk starting at frame 2500 and lasting 500 frames needs to be stored to the file and that at that time the `SpikeTimes` datasets contains 1000 items from previous data chunks. Then the new row in the TOC will contain 2500 in FRAME START column, 3000 in the FRAME END column and 1000 in the column used by the `SpikeTimes` dataset (i.e., `SpikeTOC`; see later).

Inside Root/TOC		Inside WellGroup's TOC			
FRAME START (included)	FRAME END (excluded)	DATASET_1 POS	DATASET_2 POS	...	DATASET_K POS
FS (1)	FE (1)	DP_1 (1)	DP_2 (1)	...	DP_K (1)
FS (2)	FE (2)	DP_1 (2)	DP_2 (2)	...	DP_K (2)
...	...	...	...	...	...
FS (N)	FE (N)	DP_1 (N)	DP_2 (N)	...	DP_K (N)

# BRW-File

## STRUCTURE

Root Version	400
Well Group Version	100



## CONTAINED OBJECTS

There can only be one type of raw data in a BRW-file at any time. Thus, it will always contain either Raw, WaveletBasedRaw or EventBasedSparseRaw dataset, and the corresponding TOC dataset.

- **Raw**: a 1-dimensional dataset of Bytes containing recorded data when no compression is applied.
- **RawTOC**: a 1-dimensional dataset of 64-bit Integers containing the position inside the Raw dataset of the first raw value recorded for each data chunk when no compression is applied.
- **WaveletBasedEncodedRaw**: a 1-dimensional dataset of 16-bit Integers containing recorded data when wavelet compression is applied.
- **WaveletBasedEncodedRawTOC**: a 1-dimensional dataset of 64-bit Integers containing the position inside the WaveletBasedEncodedRaw of the first compressed value recorded for each data chunk when wavelet compression is applied. It has two 32-bit Integer attributes **CompressionLevel** and **DataChunkLength**, where the former indicates the amount of compression applied, and the latter indicates the number of samples represented by the coefficients in each chunk.
- **EventsBasedSparseRaw**: a 1-dimensional dataset of Bytes containing recorded data when events-based compression (currently, this compression corresponds to the Noise Blanking compression; see also BrainWave 5 user guide) is applied.

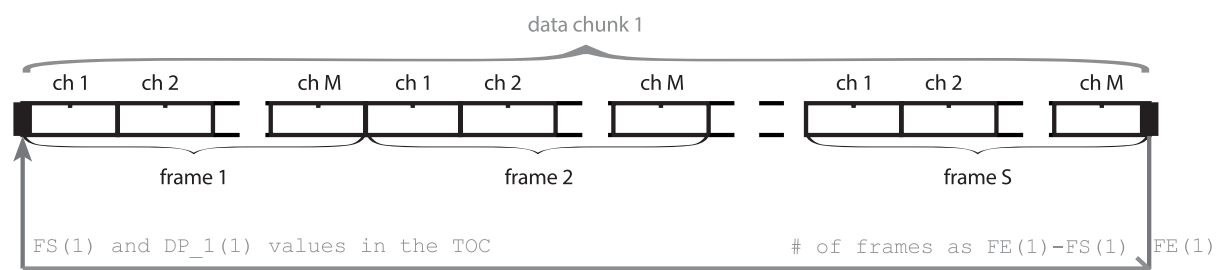
- **EventsBasedSparseRawTOC**: a 1-dimensional dataset of 64-bit Integers containing the position inside the EventsBasedSparseRaw dataset of the first compressed value recorded for each data chunk when events-based compression is applied.
- **NoiseMean**: a 1-dimensional dataset of 32-bit floating-point values containing the measured mean values for each channel and for each data chunk. Values are referred to digital sample values that need to be converted to analog values.
- **NoiseStdDev**: a 1-dimensional dataset of 32-bit floating-point values containing the measured standard deviation values for each channel and for each data chunk. Values are referred to digital sample values that need to be converted to analog values.
- **NoiseTOC**: a 1-dimensional dataset of 64-bit Integers containing the position inside noise-related datasets (e.g., NoiseMean NoiseStdDev) of the first noise value recorded for each data chunk.
- **NoiseChIdxs**: a 1-dimensional dataset of 32-bit Integers containing the linear indexes of all channels whose noise values have been considered valid. During recording, some channels showing noise values outside the normal distribution might be considered outliers and ignored for the analysis.
- **StoredChIdxs**: a 1-dimensional dataset of 32-bit Integers containing the linear indexes of all BioSPU-chip channels that have been recorded.

## RAW

The Raw dataset can contain either one single Recording Interval (possibly divided into multiple contiguous data chunks) or multiple Recording Intervals. In the latter case, the recorded intervals are saved from the stream coming from the CEI Plate according to the recording conditions, like for instance user-defined intervals, or protocol-defined intervals.

The Raw array consists of a series of Bytes that define all the frames inside the data chunks. A frame is indivisible and cannot be partially recorded. As illustrated in the following picture, for each data chunk a range of values is stored. The beginning of each chunk is identified by  $FS(i)$  that is stored in the Root TOC and represents the starting frame number of the chunk, and by  $DP_K(i)$  that is stored in the RawTOC and marks the position in number of elements (in this case Bytes) inside the Raw dataset.

Each range consists of  $M \times S$  values, where  $M$  is the number of channels recorded, and  $S$  is the number of samples (frames) per chunk per channel. Each sample is encoded in digital values and converted into the corresponding analogue value (see [Samples](#)).



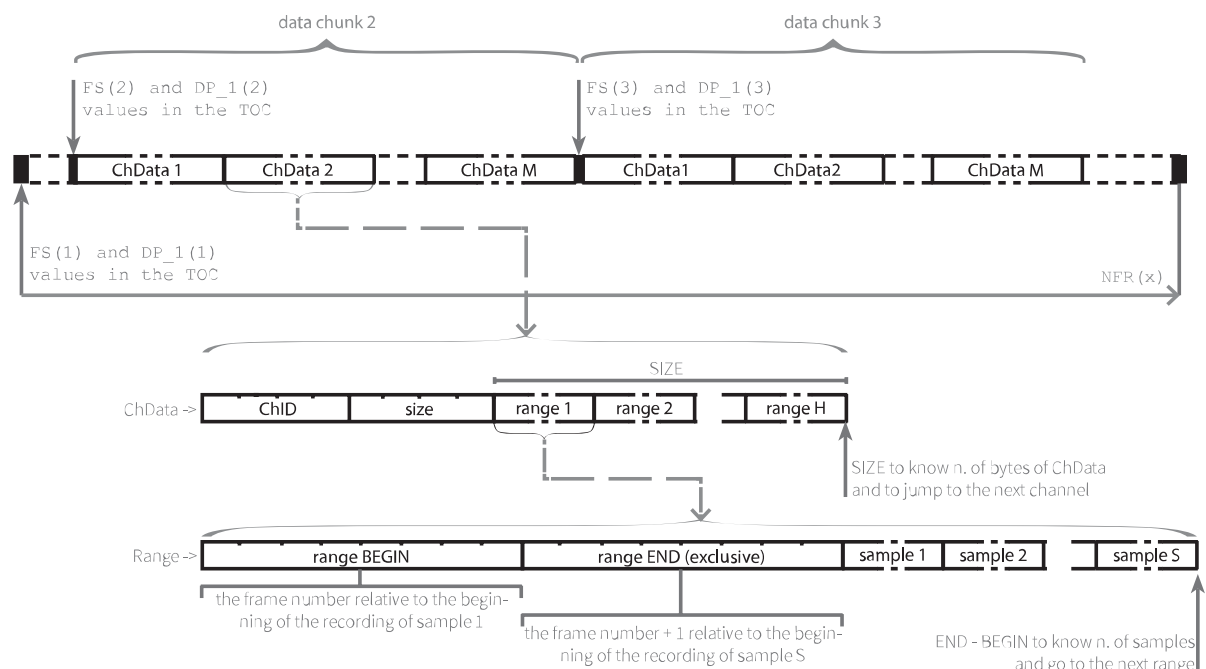
## EVENT BASED SPARSE RAW

This type of encoding defines the ranges of data (raw ranges, not to be confused with Recording Intervals nor with data chunks) that are saved from the stream according to the identified events, which may be for instance spike or field potential events. The saved raw ranges depend on the compression parameters and typically consist of samples around the actual timestamp of the identified event. The remaining portions of the stream, i.e., those where no events are found, are instead discarded.

The `EventsBasedSparseRaw` array consists of a series of bytes that define the raw ranges saved for each channel. Referring to the below picture, at each position in the array defined in the TOC, a series of `ChData` elements will be found for each channel whose data has been stored.

A `ChData` consists of a small header of 8 bytes, where the first 4 bytes encode the channel tag with a 32-bit integer, i.e. the linear index of the channel in the BioSPU-microchip, and the following 4 bytes (32-bit integer) represent the size of the `ChData` (without including the header itself). Size can be used to jump to the following `ChData`. The following bytes in the `ChData` element encodes the raw ranges saved for the channel as `Range` elements.

Each `Range` then consists of a small header with two 64-bit integers that indicate the begin of the range in number of frames (relative to the beginning of the recording) and the end (exclusive) of the range in number of frames. The difference between end and begin gives the number of samples that will follow in the `Range` element. Each sample represents the recorded biosignal data as digital values. Conversion from recorded sample index to time information and from digital values to analogue values can be performed by using the recording variables.



The following is the Python function to decode the event-based sparse raw data and transform it into raw data. The decoding takes place for all the channels of a given well ID (“Well\_A1” defines the single well in a HD-MEA or the well at first row and first column in a CorePlate™) and a given time frame interval. The input data is a dictionary where the keys are the recorded channel

indexes `StoredChIdxs` (see details in **Contained Objects**) and the values an array initialized with `numFrames` zeros for each key. Any `Range` found within the requested frame interval takes the place of the zeros at the `Range` relative time location. The returned data list contains digital samples that can be converted into analog values as detailed in **Samples**.

```
def DecodeEventBasedRawData(file, data, wellID, startFrame, numFrames):
    # collect the TOCs
    toc = np.array(file['TOC'])
    eventsToc = np.array(file[wellID + '/EventsBasedSparseRawTOC'])

    # from the given start position and duration in frames, localize the corresponding event positions
    # using the TOC
    tocStartIdx = np.searchsorted(toc[:, 1], startFrame)
    tocEndIdx = min(np.searchsorted(toc[:, 1], startFrame + numFrames, side='right')
                    + 1, len(toc) - 1)

    eventsStartPosition = eventsToc[tocStartIdx]
    eventsEndPosition = eventsToc[tocEndIdx]

    # decode all data for the given well ID and time interval
    binaryData = file[wellID + '/EventsBasedSparseRaw'][eventsStartPosition:eventsEndPosition]
    binaryDataLength = len(binaryData)

    pos = 0
    while pos < binaryDataLength:
        chIdx = int.from_bytes(binaryData[pos:pos + 4], byteorder='little', signed=True)
        pos += 4

        chDataLength = int.from_bytes(binaryData[pos:pos + 4], byteorder='little', signed=True)
        pos += 4

        chDataPos = pos
        while pos < chDataPos + chDataLength:
            fromInclusive = int.from_bytes(binaryData[pos:pos + 8], byteorder='little', signed=True)
            pos += 8

            toExclusive = int.from_bytes(binaryData[pos:pos + 8], byteorder='little', signed=True)
            pos += 8

            rangeDataPos = pos
            for j in range(fromInclusive, toExclusive):
                if j >= startFrame + numFrames:
                    break
                if j >= startFrame:
                    data[chIdx][j - startFrame] = int.from_bytes(
                        binaryData[rangeDataPos:rangeDataPos + 2], byteorder='little', signed=True)

                    rangeDataPos += 2

            pos += (toExclusive - fromInclusive) * 2
```

In the previous code, the gaps between two consecutive `Ranges` are filled with zeros. To instead have those gaps filled with Gaussian noise, the channels' mean and standard deviation values captured during the recording can be used. The following function initializes the data list with such synthetically recreated noise.

```
def GenerateSyntheticNoise(file, data, wellID, startFrame, numFrames):
    # collect the TOCs
    toc = np.array(file['TOC'])
    noiseToc = np.array(file[wellID + '/NoiseTOC'])

    # from the given start position in frames, localize the corresponding noise positions
    # using the TOC
    tocStartIdx = np.searchsorted(toc[:, 1], startFrame)
    noiseStartPosition = noiseToc[tocStartIdx]
    noiseEndPosition = noiseStartPosition

    for i in range(tocStartIdx + 1, len(noiseToc)):
        nextPosition = noiseToc[i]
        if nextPosition > noiseStartPosition:
```

```

        noiseEndPosition = nextPosition
        break
    if noiseEndPosition == noiseStartPosition:
        for i in range(tocStartIdx - 1, 0, -1):
            previousPosition = noiseToc[i]
            if previousPosition < noiseStartPosition:
                noiseEndPosition = noiseStartPosition
                noiseStartPosition = previousPosition
                break

    # obtain the noise info at the start position
    noiseChIdxs = file[wellID + '/NoiseChIdxs'][noiseStartPosition:noiseEndPosition]
    noiseMean = file[wellID + '/NoiseMean'][noiseStartPosition:noiseEndPosition]
    noiseStdDev = file[wellID + '/NoiseStdDev'][noiseStartPosition:noiseEndPosition]
    noiseLength = noiseEndPosition - noiseStartPosition

    noiseInfo = {}
    meanCollection = []
    stdDevCollection = []

    for i in range(1, noiseLength):
        noiseInfo[noiseChIdxs[i]] = [noiseMean[i], noiseStdDev[i]]
        meanCollection.append(noiseMean[i])
        stdDevCollection.append(noiseStdDev[i])

    # calculate the median mean and standard deviation of all channels to be used for
    # invalid channels
    dataMean = np.median(meanCollection)
    dataStdDev = np.median(stdDevCollection)

    # fill with Gaussian noise
    for chIdx in data:
        if chIdx in noiseInfo:
            data[chIdx] = np.array(np.random.normal(noiseInfo[chIdx][0], noiseInfo[chIdx][1],
                                                    numFrames), dtype=np.int16)
        else:
            data[chIdx] = np.array(np.random.normal(dataMean, dataStdDev, numFrames),
                                   dtype=np.int16)

```

Finally, the following Python script makes use of the two previous functions and illustrates the entire procedure of reading and plotting from a file an event-based sparse raw data time interval at well A1.

```

import numpy
import numpy as np
import matplotlib.pyplot as plt
import h5py

# variables
fileDirectory = 'D:\\'
fileName = 'testFile.brw'
wellID = 'Well_A1'
useSyntheticNoise = True

dataStartPositionSec = 0
dataDurationSec = 3

# open the BRW file
file = h5py.File(fileDirectory + fileName, 'r')

# collect experiment information
samplingRate = file.attrs['SamplingRate']
chIdxs = np.array(file[wellID + '/StoredChIdxs'])

# convert the requested time interval from seconds to frames
startFrame = int(dataStartPositionSec * samplingRate)
numFrames = int(dataDurationSec * samplingRate)

# initialize an empty (fill with zeros) data collection
data = {}
for chIdx in chIdxs:
    data[chIdx] = np.zeros(numFrames, dtype=np.int16)

```



```

# fill the data collection with Gaussian noise if requested
if useSyntheticNoise:
    GenerateSyntheticNoise(file, data, wellID, startFrame)

# fill the data collection with the decoded event based sparse raw data
DecodeEventBasedRawData(file, data, wellID, startFrame, numFrames)

# close the file
file.close()

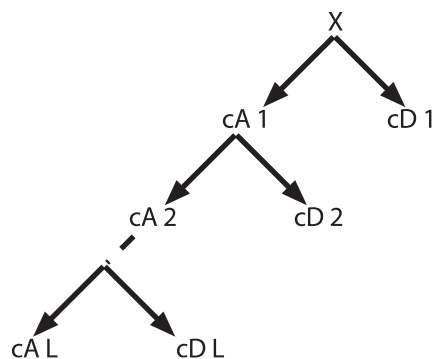
# visualize the decoded raw signal from the first channel
x = np.arange(startFrame, startFrame + numFrames, 1) / samplingRate
y = np.fromiter(data[chIdxs[0]], float)

plt.figure()
plt.plot(x, y, color="blue")
plt.title('Raw Signal')
plt.xlabel('(sec)')
plt.ylabel('(ADC Count)')
plt.show()

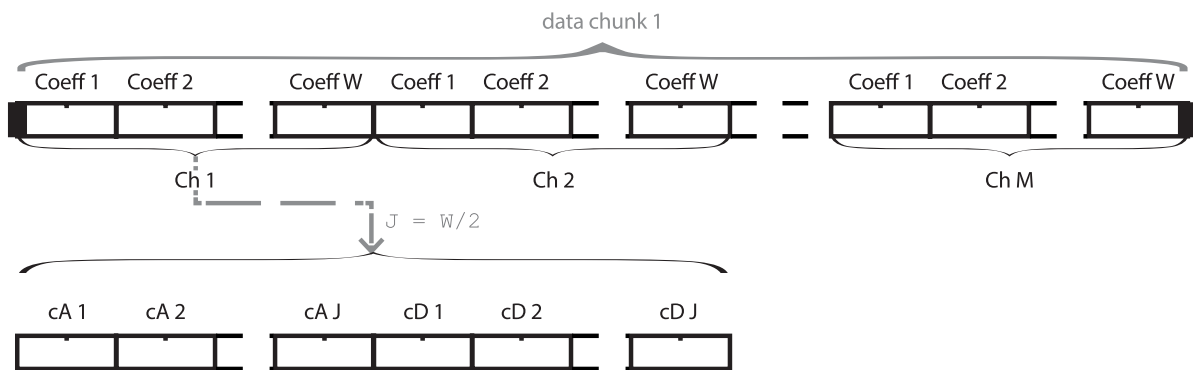
```

## WAVELET BASED ENCODED RAW

With this mode, the raw data is compressed by storing the last approximate ( $c_{A_L}$ ) and detail ( $c_{D_L}$ ) coefficients of its multi-level Discrete Wavelet Transform decomposition. The stored coefficients can be used to perform data reconstruction.



The WaveletBasedEncodedRaw dataset consists of multiple contiguous data chunks where each one stores an array of wavelet coefficients for each recorded channel. The data chunk consists of  $M \times W$  values, where  $M$  is the number of recorded channels, and  $W = \text{ceiling}(\text{DataChunkLength} / \text{pow}(2, \text{CompressionLevel})) * 2$  is the number of wavelet coefficients. Each array of  $W$  coefficients contains the required information to reconstruct a corresponding array of  $\text{DataChunkLength}$  digital samples (frames) for a given channel. The  $W$  coefficients contains both the array of  $c_A$  approximation coefficients (left half) and the  $c_D$  detail coefficients (right half) of the last multi-level wavelet decomposition.



The position of each data chunk inside `WaveletBasedEncodedRaw` is defined by the combination of the Root TOC and the `WaveletBasedEncodedRawTOC` dataset.

### Data Reconstruction

Let  $X = \text{idwt}(c_A, c_D)$  be an Inverse Discrete Wavelet Transform function that returns the wavelet reconstruction  $X$  based on the  $c_A$  and  $c_D$  coefficients using the **Symlets 7** wavelet and the **Periodization** extension mode for dealing with the problem of border distortion. Moreover, let  $C = [c_1, c_2, \dots, c_W]$  be the coefficient array and `zeros(j)` a function that returns an array of  $j$  zeros values. By performing the following multi-level reconstruction cycle,

```

j = W / 2
X = idwt(C[1:j], C[j:W])
j = j*2
for i = 2:CompressionLevel
    X = idwt(X[1:j], zeros(j))
    j = j*2
end

```

the  $X$  will contain the reconstructed signal samples array of dimension `DataChunkLength`.

Here below an example in Python to reconstruct the signal from Well A1 (either a HD-MEA or the well at first row and first column in a `CorePlate™`):

```

import numpy as np
import math
import matplotlib.pyplot as plt
import h5py
import pywt

# variables
fileDirectory = 'D:\\'
fileName = 'fileName.brw'
chIdx = 1

# open the BRW file
file = h5py.File(fileDirectory + fileName, 'r')

# collect experiment information
samplingRate = file.attrs['SamplingRate']
nChannels = len(file['Well_A1/StoredChIdxs'])
coefsTotalLength = len(file['Well_A1/WaveletBasedEncodedRaw'])
compressionLevel = file['Well_A1/WaveletBasedEncodedRaw'].attrs['CompressionLevel']
framesChunkLength = file['Well_A1/WaveletBasedEncodedRaw'].attrs['DataChunkLength']
coefsChunkLength = math.ceil(framesChunkLength / pow(2, compressionLevel)) * 2

# reconstruct all data for a given channel index
data = []
coefsPosition = chIdx * coefsChunkLength

```

```

while coefsPosition < coefsTotalLength:
    coefs = file['Well_A1/WaveletBasedEncodedRaw'][coefsPosition:coefsPosition + coefsChunkLength]
    length = int(len(coefs) / 2)

    frames = pywt.idwt(coefs[:length], coefs[length:], 'sym7', 'periodization')
    length *= 2

    for i in range(1, compressionLevel):
        frames = pywt.idwt(frames[:length], None, 'sym7', 'periodization')
        length *= 2

    data.extend(frames)
    coefsPosition += coefsChunkLength * nChannels

# close the file
file.close()

# visualize the reconstructed raw signal
x = np.arange(0, len(data), 1) / samplingRate
y = np.fromiter(data, float)

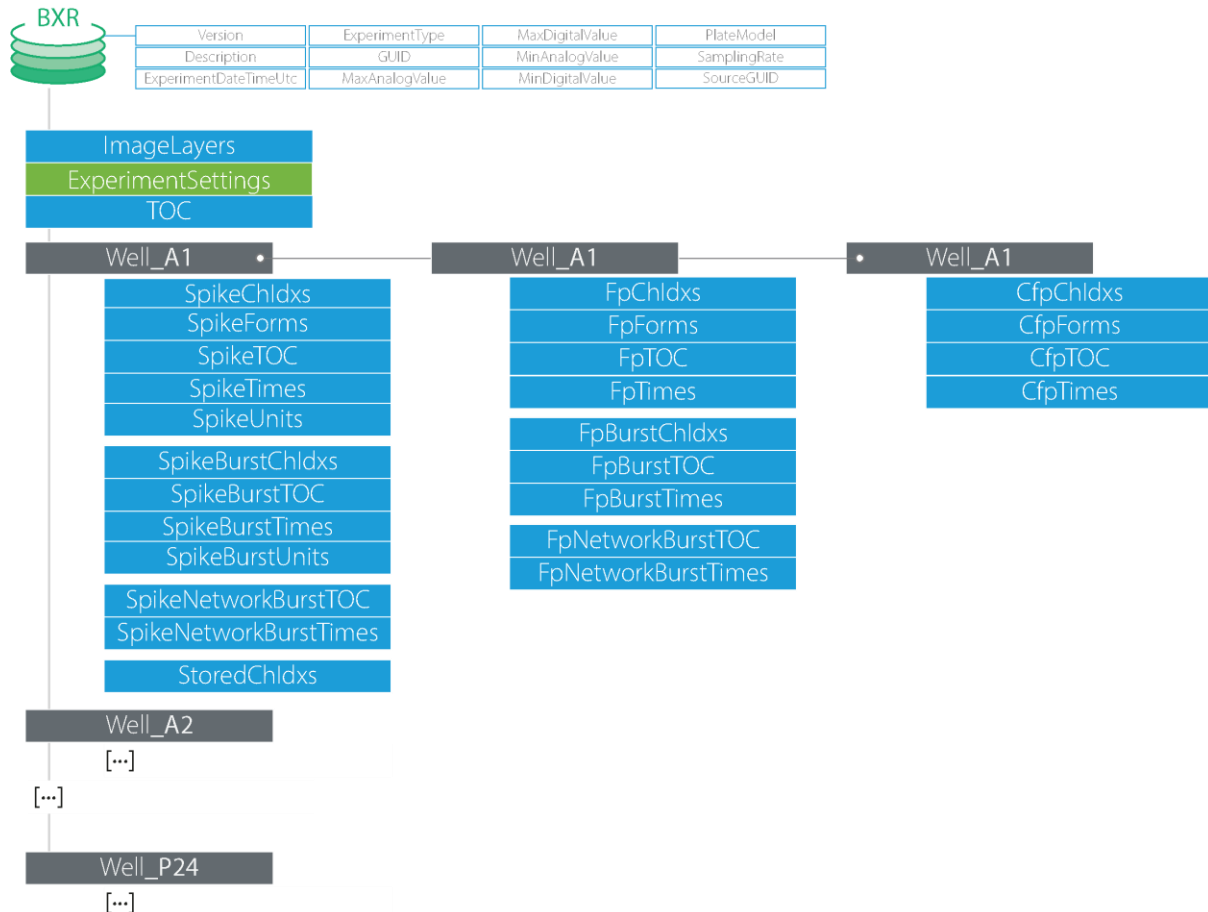
plt.figure()
plt.plot(x, y, color="blue")
plt.title('Reconstructed Raw Signal')
plt.xlabel('(sec)')
plt.ylabel('(ADC Count)')
plt.show()

```

# BXR-File

## STRUCTURE

Root Version	300
Current Version	301
Well Group Version	101



## CONTAINED OBJECTS

Each kind of event recorded by Brainwave is defined by a set of properties. For example, a spike can be defined through the time at which it was recorded, the channel it was recorded on, and its waveform. This information is split over multiple datasets, one for each property, but all datasets share the same number of elements so that a given event is defined by all the properties found at the same position in the corresponding datasets.

For each type of event a TOC is also defined, which indicates for each recorded data chunk the position of the first event recorded.

Here the list of all available datasets:

- **SpikeChIdxs:** a 1-dimensional array of 32-bit Integers, representing for each detected spike the linear index of the channel it has been recorded on.
- **SpikeForms:** a 1-dimensional array of 16-bit Integers of length  $M = N \times W$ , where  $N$  is the number of spikes detected, and  $W$  is the number of frames stored for each waveform. It

contains the waveforms for each detected spike, collapsed into one dimension. It has a 32-bit Integer attribute `WaveLength` indicating the length, in frames, of the waveforms ( $W$ ). From Root Version 301, it has a 32-bit Integer attribute `WaveTimeOffset` indicating the position, in frames, of the detected peak in the wave, which corresponds to the action potential's peak. To get the position of the waveform (event's property) of a given spike, the value found in `SpikeTOC` must be multiplied by  $W$ .

- `SpikeTOC`: a 1-dimensional array of 64-bit Integers, containing the position inside spike datasets (e.g., `SpikeChIdxs`, `SpikeTimes`) of the first spike detected for each data chunk.
- `SpikeTimes`: a 1-dimensional array of 64-bit Integers, representing the time instant, in number of frames, in which each spike has been detected.
- `SpikeUnits`: a 1-dimensional array of 32-bit Integers, representing the Unit Identifier for each detected spike. It is defined only if Spike Sorting has been performed.
- `SpikeBurstChIdxs`: a 1-dimensional array of 32-bit Integers, representing for each detected spike burst the linear index of the channel it has been recorded on.
- `SpikeBurstTOC`: a 1-dimensional array of 64-bit Integers, containing the position inside spike burst datasets (e.g., `SpikeBurstChIdxs`, `SpikeBurstTimes`) of the first spike burst detected for each data chunk.
- `SpikeBurstTimes`: a 1-dimensional array of 64-bit Integers, representing the time instant, in frames, in which each spike burst has been detected.
- `SpikeBurstUnits`: a 1-dimensional array of 32-bit Integers, representing the Unit Identifier for each detected spike burst. It is defined only if Spike Sorting has been performed.
- `SpikeNetworkBurstTOC`: a 1-dimensional array of 64-bit Integers, containing the position inside spike network burst datasets (e.g., `SpikeNetworkBurstChIdxs`, `SpikeNetworkBurstTimes`) of the first spike network burst detected for each data chunk.
- `SpikeNetworkBurstTimes`: a 1-dimensional array of 64-bit Integers, representing the time instant, in frames, in which each spike network burst has been detected.
- `FpChIdxs`: a 1-dimensional array of 32-bit Integers, representing for each detected field potential the linear index of the channel it has been recorded on.
- `FpForms`: a 1-dimensional array of 16-bit Integers of length  $M = N \times W$ , where  $N$  is the number of field potentials detected, and  $W$  is the number of frames stored for each waveform. It contains the waveforms for each detected field potential, collapsed into one dimension. It has a 32-bit Integer attribute `WaveLength` indicating the length, in frames, of the waveforms ( $W$ ). As field potentials can have different waveform length, waves shorter than `WaveLength` frames are 0-padded. To get the position of the waveform (event's property) of a given field potential, the value found in `FpTOC` must be multiplied by  $W$ .
- `FpTOC`: a 1-dimensional array of 64-bit Integers, containing the position inside field potential datasets (e.g., `FpChIdxs`, `FpTimes`) of the first field potential detected for each data chunk.
- `FpTimes`: a 1-dimensional array of 64-bit Integers, representing the time instant, in frames, in which each field potential has been detected.
- `FpBurstChIdxs`: a 1-dimensional array of 32-bit Integer values, representing for each detected field potential burst the linear index of the channel it has been recorded on.

- **FpBurstTOC**: a 1-dimensional array of 64-bit Integer values, representing the position inside field potential burst datasets (e.g., `FpBurstChIdxs`, `FpBurstTimes`) of the first field potential burst event type at the beginning of each data.
- **FpBurstTimes**: a 1-dimensional array of 64-bit Integers, representing the time instant, in frames, in which each field potential burst has been detected.
- **FpNetworkBurstTOC**: a 1-dimensional array of 64-bit Integers, containing the position inside field potential network burst datasets (`FpNetworkBurstChIdxs`, `FpNetworkBurstTimes`) of the first field potential network burst detected for each data chunk.
- **FpNetworkBurstTimes**: a 1-dimensional array of 64-bit Integers, representing the time instant, in frames, in which each field potential network burst has been recorded.
- **CfpTOC**: a 1-dimensional array of 64-bit Integers, containing the position inside cardiac field potential datasets (e.g., `CfpChIdxs`, `CfpTimes`) of the first cardiac field potential detected for each data chunk.
- **CfpTimes**: a 4-dimensional array of 64-bit Integers, representing the time instants, in frames, of the Q, R, S and T points of each detected cardiac field potential. A value of -1 means that it was not possible to detect such point for that event.
- **CfpChIdxs**: a 1-dimensional array of 32-bit Integer values, representing for each detected cardiac field potentials the linear index of the channel it has been recorded on.
- **CfpForms**: a 1-dimensional array of 16-bit Integers of length  $M = N \times W$ , where  $N$  is the number of cardiac field potentials detected, and  $W$  is the number of frames stored for each waveform. It contains the waveforms for each detected field potential, collapsed into one dimension. It has a 32-bit Integer attribute `WaveLength` indicating the length, in frames, of the waveforms ( $W$ ). It also has a 32-bit Integer attribute `WaveTimeOffset` indicating the position, in frames, of the detected peak in the wave, which corresponds to R if positive threshold has been used, S otherwise. To get the position of the waveform (event's property) of a given cardiac field potential, the value found in `CfpTOC` must be multiplied by  $W$ .

# BCMP-File

The BCMP format contains links and instructions to combine together multiple BRW-files or BXR-files. For instance, this is useful when multiple BXR-files were obtained from the same CEI Plate at different time points (e.g. at different days). Combining the files together allows you to see the data trend over time for your experiment. Conceptually, a BCMP-file is created starting from BXR-files, hence a BCMP-file is associated and linked with multiple BXR-files.

Root Version	100
--------------	-----

## CONTAINED OBJECTS

BCMP-Files contain a JSON-encoded object that stores information on the file itself and on the other files that are linked by the BCMP-File (file paths and their GUID).